Вопросы к коллоквиуму для студентов II-го курса

по дисциплине «Скриптовые языки программирования»

**Тема 1. Основы JavaScript**

**Тема 2. Структуры данных**

**Тема 3. Функции и замыкания**

**Тема 4. Методы объектов и контекст вызова**

**Тема 5. ООП в функциональном и прототипном стиле**

### 1. \*\*Переменные. Имена переменных. Константы.\*\*

Переменные используются для хранения данных, которые можно переиспользовать в программе. В JavaScript переменные объявляются с помощью ключевых слов `let`, `const` и `var`.

- `let` позволяет объявлять переменные, которые можно изменить в будущем.

- `const` используется для объявления постоянных переменных, значения которых нельзя изменить после инициализации.

Имена переменных должны начинаться с буквы, символа подчеркивания `\_` или знака доллара `$` и не должны совпадать с зарезервированными словами (например, `let`, `return`, `function` и т.д.).

let имя = "Алиса";

const возраст = 25;

### 2. \*\*Типы данных. Оператор `typeof`.\*\*

JavaScript поддерживает несколько основных типов данных:

- \*\*Число (number)\*\*: целые и вещественные числа.

- \*\*Строка (string)\*\*: текстовые данные, заключённые в кавычки.

- \*\*Булев (boolean)\*\*: логический тип, принимающий значения `true` или `false`.

- \*\*Объект (object)\*\*: структура данных, которая может содержать другие значения.

- \*\*Undefined\*\*: значение переменной, которая была объявлена, но не инициализирована.

- \*\*Null\*\*: намеренное отсутствие значения.

- \*\*Symbol\*\*: уникальные идентификаторы.

- \*\*BigInt\*\*: для представления целых чисел произвольной длины.

Оператор `typeof` позволяет узнать тип данных переменной.

let x = 42;

console.log(typeof x); // "number"

### 3. \*\*Преобразование типов.\*\*

JavaScript автоматически преобразует типы данных в некоторых ситуациях. Это называется \*\*неявным преобразованием\*\*. Также возможны \*\*явные преобразования\*\* с помощью специальных методов, таких как `Number()`, `String()`, `Boolean()`

let num = "123";

console.log(Number(num)); // 123 (преобразование строки в число)

console.log(Boolean(0)); // false (преобразование числа в логическое значение)

### 4. \*\*Операторы. Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов.\*\*

Операторы в JavaScript можно разделить на:

- \*\*Унарные\*\* операторы работают с одним операндом. Пример: `-x` (смена знака числа).

- \*\*Бинарные\*\* операторы работают с двумя операндами. Пример: `x + y`.

Операторы также имеют приоритет, который определяет порядок их выполнения в выражении.

let x = 5;

console.log(-x); // -5 (унарный оператор)

console.log(5 + 3); // 8 (бинарный оператор)

### 5. \*\*Логические операторы.\*\*

Логические операторы используются для выполнения операций с булевыми значениями:

- `&&` — логическое И (возвращает true, если оба операнда true).

- `||` — логическое ИЛИ (возвращает true, если хотя бы один операнд true).

- `!` — логическое НЕ (инвертирует значение).

let isAdult = true;

let hasPermission = false;

console.log(isAdult && hasPermission); // false

### 6. \*\*Функции (Function Declaration). Параметры по умолчанию.\*\*

Функции — это блоки кода, которые можно вызывать многократно. Декларация функций позволяет создать именованную функцию, которую можно вызвать по имени.

Параметры по умолчанию позволяют задавать значение параметра, если аргумент не передан.

function greet(name = "Гость") {

console.log("Привет, " + name);

}

greet(); // "Привет, Гость"

### 7. \*\*Функциональные выражения (Function Expression) и функции-стрелки.\*\*

Функциональные выражения создают функцию и присваивают её переменной. Функции-стрелки (`=>`) — это сокращённый синтаксис для создания функций, они не имеют собственного контекста `this`.

const sayHello = function(name) {

console.log("Привет, " + name);

};

const sayHi = (name) => console.log("Привет, " + name);

### 8. \*\*Числа. Способы записи числа. Системы счисления. Методы типа Number.\*\*

JavaScript поддерживает разные способы записи чисел, включая десятичные, шестнадцатеричные (`0x`), восьмеричные (`0o`) и двоичные (`0b`). Методы типа `Number` включают `toFixed()`, `isNaN()`, и т.д.

let num = 123.456;

console.log(num.toFixed(2)); // "123.46"

### 9. \*\*Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк.\*\*

Строки могут быть преобразованы и изменены с помощью методов, таких как `toUpperCase()` (приведение к верхнему регистру) и `includes()` (поиск подстроки).

let text = "hello";

console.log(text.toUpperCase()); // "HELLO"

console.log(text.includes("ell")); // true

### 10. \*\*Массивы.\*\*

Массивы — это упорядоченные коллекции данных. Каждый элемент имеет индекс, начиная с 0.

let fruits = ["apple", "banana", "cherry"];

### 11. \*\*Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.\*\*

Методы массивов включают:

- `push()` — добавляет элемент в конец.

- `pop()` — удаляет последний элемент.

- `concat()` — объединяет массивы.

let arr = [1, 2, 3];

arr.push(4); // [1, 2, 3, 4]

### 12. \*\*Методы перебора и преобразование массива.\*\*

Методы перебора массива, такие как `map()` и `filter()`, позволяют выполнять действия с каждым элементом.

let numbers = [1, 2, 3];

let doubled = numbers.map(n => n \* 2); // [2, 4, 6]

### 13. \*\*Объекты. Литералы и свойства.\*\*

Объекты — это коллекции пар "ключ-значение", которые можно создавать с помощью объектных литералов.

let person = { name: "Алиса", age: 25 };

### 14. \*\*Копирование, клонирование, сравнение, объединение объектов.\*\*

Копирование объектов можно выполнять с помощью `Object.assign()` или оператора расширения `{...}`.

### 15. \*\*Коллекции Set, WeakSet.\*\*

`Set` — это коллекция уникальных значений, `WeakSet` — коллекция объектов с "слабыми" ссылками.

let uniqueValues = new Set([1, 2, 3, 3]);

### 16. \*\*Коллекции Map, WeakMap.\*\*

`Map` — коллекция пар "ключ-значение", `WeakMap` позволяет использовать только объекты в качестве ключей.

let map = new Map();

map.set("key", "value");

### 17. \*\*Деструктурирующее присваивание. Вложенная деструктуризация.\*\*

Деструктуризация позволяет извлекать значения из массивов и объектов и присваивать их переменным.

let [a, b] = [1, 2];

let { name, age } = person;

### 18. \*\*Глобальный объект. Создание функции с помощью конструктора (new Function).\*\*

Глобальный объект включает встроенные функции и переменные, такие как `Math` и `console`. `new Function()` создаёт функции динамически.

let func = new Function("a", "b", "return a + b");

console.log(func(1, 2)); // 3

### 19. \*\*Лексическое окружение (LexicalEnvironment). Замыкание.\*\*

Замыкания позволяют функции запоминать и получать доступ к переменным своей внешней функции, даже если внешняя функция завершена.

function outer() {

let x = 10;

return function inner() {

console.log(x); // 10

};

}

outer()();

### 20. \*\*Объект функции. Именованное функциональное выражение (Named Function Expression).\*\*

Именованные функциональные выражения дают функции имя, что полезно для рекурсии и отладки.

const factorial = function fact(n) {

return n <= 1 ? 1 : n \* fact(n - 1);

};

### 21. \*\*Остаточные параметры и оператор расширения.\*\*

Остаточные параметры `...args` собирают все аргументы функции в массив. Оператор расширения `...` используется для развёртывания массивов и объектов.

function sum(...args) {

return args.reduce((a, b) => a + b, 0);

}

### 22. \*\*Каррирование и частичное применение функции.\*\*

Каррирование превращает функцию, принимающую несколько аргументов, в цепочку функций с одним аргументом.

function multiply(a) {

return function(b) {

return a \* b;

};

}

### 23. \*\*Генераторы. Функции-генераторы. Перебор объектов-генераторов.\*\*

\*\*Функции-генераторы\*\* — это функции, которые могут приостанавливать своё выполнение и возобновлять его позже. Такие функции определяются с помощью `function\*` и возвращают специальный объект — генератор, который позволяет получать значения поочерёдно. Для управления потоком генератора используются команды `yield` (приостановка и возврат значения) и `next()` (продолжение выполнения).

function\* myGenerator() {

yield 1;

yield 2;

yield 3;

}

const gen = myGenerator();

console.log(gen.next().value); // 1

console.log(gen.next().value); // 2

### 24. \*\*Методы объектов, `this`. Оператор опциональной последовательности.\*

Методы — это функции, которые находятся в объекте и работают с его данными. `this` в методах указывает на текущий объект.

\*\*Оператор опциональной последовательности\*\* `?.` позволяет обращаться к свойствам или методам объекта, даже если один из промежуточных объектов не определён, избегая ошибок.

let user = {

name: "Alice",

greet() {

console.log(`Привет, ${this.name}`);

}

};

user.greet(); // Привет, Alice

console.log(user?.profile?.age); // undefined, не вызывает ошибку

### 25. \*\*Преобразование объектов.\*\*

JavaScript автоматически преобразует объекты к строкам или числам, когда это необходимо. Можно определить, как объект должен преобразовываться, используя методы `toString()` и `valueOf()`.

let obj = {

value: 10,

toString() {

return `Объект со значением ${this.value}`;

}

};

console.log(String(obj)); // "Объект со значением 10"

### 26. \*\*Создание объектов через `new`.\*\*

Конструкторные функции позволяют создавать объекты с помощью оператора `new`. Такие функции создают экземпляры объектов с собственными свойствами и методами.

function User(name) {

this.name = name;

this.greet = function() {

console.log(`Привет, ${this.name}`);

};

}

let user = new User("Alice");

user.greet(); // Привет, Alice

### 27. \*\*Флаги и дескрипторы свойств.\*\*

Каждое свойство объекта в JavaScript имеет \*\*дескриптор\*\*, который определяет его поведение. Основные флаги дескрипторов:

- `writable`: можно ли изменить значение свойства.

- `enumerable`: можно ли перебирать свойство в цикле `for...in`.

- `configurable`: можно ли удалить или изменить флаги свойства.

let person = {};

Object.defineProperty(person, "name", {

value: "Alice",

writable: false,

enumerable: true,

configurable: true

});

### 28. \*\*Геттеры и сеттеры.\*\*

\*\*Геттеры\*\* и \*\*сеттеры\*\* — это методы для получения и установки значений свойств. Они позволяют контролировать доступ к данным.

let user = {

firstName: "Alice",

lastName: "Johnson",

get fullName() {

return `${this.firstName} ${this.lastName}`;

},

set fullName(name) {

[this.firstName, this.lastName] = name.split(" ");

}

};

user.fullName = "John Doe";

console.log(user.fullName); // "John Doe"

### 29. \*\*Методы `call()`, `apply()`, `bind()`.\*\*

Эти методы управляют привязкой контекста `this` к функциям:

- `call()` вызывает функцию с указанным `this` и аргументами.

- `apply()` работает как `call()`, но принимает аргументы в виде массива.

- `bind()` возвращает новую функцию с привязанным `this`.

function greet() {

console.log(`Привет, ${this.name}`);

}

let user = { name: "Alice" };

greet.call(user); // Привет, Alice

### 30. \*\*Прототипное наследование. Собственные и унаследованные свойства. Свойство `F.prototype`.\*\*

JavaScript поддерживает \*\*прототипное наследование\*\*: объекты могут наследовать свойства и методы от других объектов. Свойство `F.prototype` у функций-конструкторов позволяет устанавливать объект-прототип для всех создаваемых объектов.

function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} издаёт звук.`);

};

let dog = new Animal("Пёс");

dog.speak(); // Пёс издаёт звук.

### 31. \*\*Классы. Class Expression. Приватные и защищённые методы и свойства.\*\*

\*\*Классы\*\* — это синтаксический сахар над прототипами, упрощающий создание объектов. Можно объявлять классы с помощью `class` и создавать защищённые свойства, обозначенные `\_` (условная защита). Также в JavaScript доступны \*\*приватные свойства\*\*, обозначенные `#`.

class User {

#password;

constructor(name, password) {

this.name = name;

this.#password = password;

}

checkPassword(password) {

return this.#password === password;

}

}

const user = new User("Alice", "secret");

console.log(user.checkPassword("secret")); // true

// console.log(user.#password); // Ошибка, так как #password приватный

### 32. \*\*Наследование классов. Переопределение методов. Статические свойства и методы. Оператор `instanceof`.\*\*

Наследование позволяет создавать классы, которые расширяют функциональность других классов. \*\*Статические методы\*\* принадлежат классу, а не экземплярам, и вызываются напрямую через класс. Оператор `instanceof` проверяет, является ли объект экземпляром конкретного класса.

class Animal {

static kingdom = "Животные";

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} издаёт звук.`);

}

}

class Dog extends Animal {

speak() {

console.log(`${this.name} лает.`);

}

}

let dog = new Dog("Пёс");

console.log(dog instanceof Animal); // true

dog.speak(); // Пёс лает.

```